SkeletalViewer Walkthrough: C++ and C#

Capturing Data with the NUI API

Abstract The SkeletalViewer sample demonstrates the use of the natural user interface (NUI) API in the Kinect™ for Windows® Software Development Kit (SDK) Beta to capture and render depth, video, and skeleton data. This document walks you through the unmanaged code (C++) and managed code (C#) versions of this sample.

Resources For a complete list of documentation for the Kinect for Windows SDK Beta, plus related reference and links to the online forums, see the beta SDK website at: http://kinectforwindows.org

In this guide

Contents

PART 1—Introduction to the SkeletalViewer Samples

Introduction

The Kinect™ for Xbox 360® sensor includes cameras that deliver depth information, color data, and skeleton tracking data. The natural user interface (NUI) API in the Kinect for Windows® Software Development Kit (SDK) Beta enables applications to access and manipulate this data.

The SkeletalViewer sample demonstrates Kinect NUI API processing that captures the depth stream, the color stream, and skeletal tracking frames and displays them on the screen. The following two versions of the SkeletalViewer sample are in the beta SDK:

- A C++ version that uses Microsoft® Direct3D® 9 and the Windows graphic display interface (GDI) to render images in a window.
- A C# version that demonstrates the use of the **Microsoft.Research.Kinect.Nui** managed interface and uses the Windows Presentation Foundation (WPF) for rendering and window management.

When you run either sample, you can see the following:

- The depth stream, which shows background in gray scale and different people in different colors. Darker colors indicate objects that are farther from the camera.
- Tracked skeletons of up to two people who have been detected within the frame.
- The color video stream, which shows the red-green-blue (RGB) image from the Kinect sensor.
- The rate at which captured frames are delivered to the application.

If moving figures are too close to the camera, unreliable or odd images might appear in the skeleton and depth views. The optimal range is 2.6 to 13.12 feet (0.8 to 4 meters). The depth and skeleton views detect people only if the entire body fits within the captured frame.

PART 2—C++ SkeletalViewer Sample

The C++ SkeletalViewer sample uses the NUI API to capture depth data, color, and skeletal tracking data and then renders the images on the screen by using Direct3D 9 and the Windows GDI. The sample implements a simple Windows graphical user interface (GUI).

The C++ SkeletalViewer requires that DirectX® SDK Version 9.0c be installed on your system. For more information on Windows and DirectX, see "Resources" in Part 4 of this document.

Program Basics

The C++ SkeletalViewer application is installed with the Kinect for Windows Software Development Kit (SDK) Beta samples in %KINECTSDK_DIR%\Samples\KinectSDKSamples.zip

The sample is implemented in the following files:

- SkeletalViewer.cpp contains the application's entry point and the main window creation and callback functions.
- NUIImpl.cpp and SkeletalViewer.h implement the *CSkeletalViewerApp* class, which captures data from the Kinect sensor and transforms it for rendering.
- DrawDevice.cpp and DrawDevice.h implement the *DrawDevice* class, which uses Direct3D to render images.

SkeletalViewer has the following basic program flow:

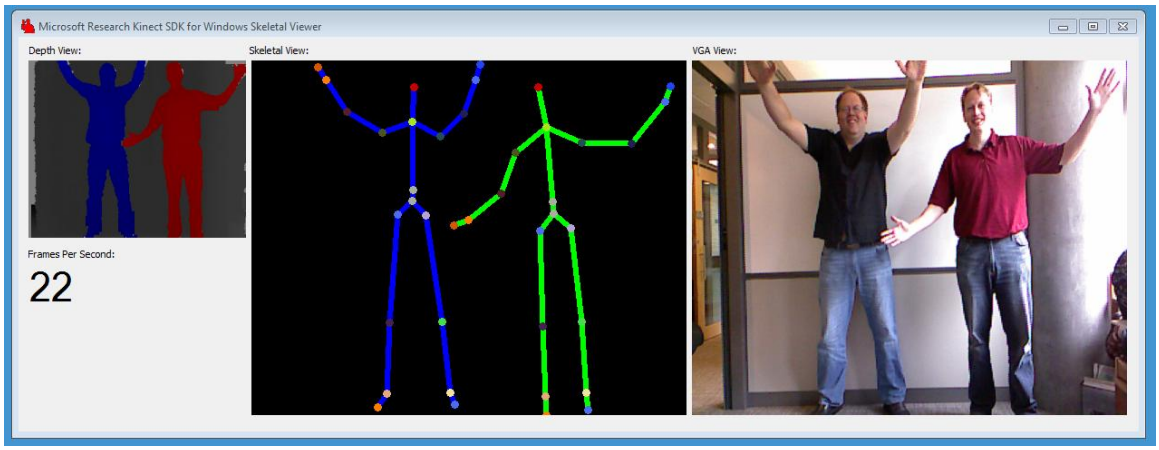
1. Create the main window in which the program displays images from the Kinect sensor.
2. Initialize the Kinect sensor, open streams for each data type, and create a thread to process the data.
3. Process sensor data and render images in the window as new data frames arrive.
4. Clean up and exit when the user closes the window.

To build and run the sample

1. In Windows Explorer, navigate to the SkeletalViewer\CPP directory.
2. Double-click the icon for the .sln (solution) file to open the file in Microsoft Visual Studio®.
3. Build the application.
4. Press CTRL+F5 to run the sample.

The solution file for the sample targets the x86 platform, because this beta SDK includes only x86 libraries.

The following illustration shows the main application window for C++ SkeletalViewer.



This document walks you through the sample and focuses on the Kinect-specific aspects of the code for data capture.

Note This document includes excerpts from the sample program, most of which have been edited for brevity and readability. In particular, most routine error correction code has been removed. For the complete code, see the SkeletalViewer sample.

Create and Manage the Main Window

The SkeletalViewer.cpp file defines the application's entry point and its window-processing callback function. If you have experience with graphical Windows applications, you will probably recognize the code in this module as boilerplate Window handling. For those who are new to such applications, this description includes links to the Microsoft Developer Network (MSDN®) for more information.

The entry point is named *_tWinmain*. When you create a new Win32® project, Microsoft Visual C++® generates the *project*.cpp source file, which includes an entry point that is named *_tWinMain* and a Window processing callback function that is named *WndProc*.

The *_tWinMain* function in SkeletalViewer creates and displays the main application window, as follows:

```
int APIENTRY _tWinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
        LPTSTR lpCmdLine, int nCmdShow)
{
MSG         msg;
WNDCLASS    wc;

// Store the instance handle
g_hInst=hInstance;
LoadString (g_hInst, IDS_APPTITLE, g_szAppTitle,
      sizeof(g_szAppTitle)/sizeof(g_szAppTitle[0]));
```

```
// Dialog custom window class
ZeroMemory (&wc,sizeof(wc));
wc.style = CS_HREDRAW | CS_VREDRAW;
wc.cbWndExtra = DLGWINDOWEXTRA;
wc.hInstance = hInstance;
wc.hCursor = LoadCursor (NULL, IDC_ARROW);
wc.hIcon = LoadIcon (hInstance, MAKEINTRESOURCE (IDI_SKELETALVIEWER));
wc.lpfnWndProc = DefDlgProc;
wc.lpszClassName = SZ_APPDLG_WINDOW_CLASS;
if (!RegisterClass(&wc))
    return (0);

// Create main application window
g_hWndApp = CreateDialogParam(g_hInst, MAKEINTRESOURCE(IDD_APP),
        NULL,(DLGPROC) CSkeletalViewerApp::WndProc,NULL);

// Show window
ShowWindow (g_hWndApp,nCmdShow);
UpdateWindow (g_hWndApp);

// Main message loop:
while( GetMessage( &msg,NULL,0,0)) {
    // If a dialog message
    if (g_hWndApp!=NULL && IsDialogMessage (g_hWndApp,&msg))
        continue;

    // otherwise do default window processing
    TranslateMessage (&msg);
    DispatchMessage (&msg);
    }
return (msg.wParam);
}
```

The window consists of a simple modeless dialog box that does not accept user input. The function follows these steps:

1. Stores the handle to the application instance and gets the name of the sample.
2. Initializes window class parameters in the *wc* variable, which is a structure of type WNDCLASS, and calls the **RegisterClass** function to register the window class.
3. Calls the **CreateDialogParam** function to create a modeless dialog box to use as the main application window, as described in Using Dialog Boxes.
4. Calls the **ShowWindow** and **UpdateWindow** functions to display and initially populate the window.
5. Loops while it retrieves and handles messages that are directed to the window. It calls the **GetMessage** function to retrieve messages and calls **TranslateMessage** and **DispatchMessage** to handle them. The loop exits when the *WndProc* callback function dispatches **PostQuitMessage.**

The *WndProc* callback function handles messages that the system sends to the window. The function is called with a handle to the application window, a message type, and additional parameters that depend upon the type of message. Because the application does not accept user input, this function is fairly straightforward, as follows:

```
LONG CALLBACK CSkeletalViewerApp::WndProc(
       HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
switch(message)
  {
  case WM_INITDIALOG:
    {
    LOGFONT lf;
    g_CSkeletalViewerApp.Nui_Zero();
    g_CSkeletalViewerApp.m_hWnd=hWnd;
    g_CSkeletalViewerApp.Nui_Init();

    GetObject((HFONT) GetStockObject(DEFAULT_GUI_FONT), sizeof(lf),&lf);
    lf.lfHeight*=4;
    g_CSkeletalViewerApp.m_hFontFPS=CreateFontIndirect(&lf);
    SendDlgItemMessage( hWnd,IDC_FPS,WM_SETFONT,
        (WPARAM) g_CSkeletalViewerApp.m_hFontFPS,0);
   }
    break;

  case WM_CLOSE:
    DestroyWindow(hWnd);
    break;

  case WM_DESTROY:
    g_CSkeletalViewerApp.Nui_UnInit();
    DeleteObject (g_CSkeletalViewerApp.m_hFontFPS);
    PostQuitMessage(0);
    break;
  }
  return (FALSE);
}
```

The window handles only the following types of messages:

- WM_INITDIALOG arrives before the window appears. In response, SkeletalViewer initializes the NUI API, begins to process data from the Kinect sensor, and sets up the font to display frames per second.
- WM_CLOSE arrives when the user clicks the box in the menu bar to close the program. SkeletalViewer simply calls the **DestroyWindow** function.
- WM_DESTROY is sent by the **DestroyWindow** function. SkeletalViewer uninitializes the NUI API, deletes the font object, and calls the **PostQuitMessage** function to inform the system that the thread is ready to terminate.

Initialize NUI API

When the WM_INITDIALOG message arrives, the *WndProc* callback function initializes the main class object, g_CSkeletalViewerApp , which is an object of type *CSkeletalViewerApp*. The *WndProc* callback function calls the *Nui_Zero* method on the g_CSkeletalViewerApp object and saves the application window handle in the object. It then calls *g_CSkeletalViewerApp::Nui_Init*. You can find this code in the NuiImpl.cpp source file. *Nui_Init* does most of the required work to initialize the Kinect sensor and begin streaming data, as follows:

- Creates an event that is associated with each type of data. The program waits on an event to determine what action to take.
- Initializes structures for skeleton rendering.
- Initializes structures for image rendering.
- Initializes the Kinect sensor.
- Opens streams for depth and color frames.
- Creates the Kinect sensor data processing thread.

The following sections walk through the *Nui_Init* code that performs these tasks.

Create Events

When a frame from the Kinect sensor is ready for the application, the runtime signals an application-supplied event. *CSkeletalViewerApp::Nui_Init* creates an event for each of the three types of data by calling the Windows **CreateEvent** function, as follows:

```
m_hNextDepthFrameEvent = CreateEvent( NULL, TRUE, FALSE, NULL );
m_hNextVideoFrameEvent = CreateEvent( NULL, TRUE, FALSE, NULL );
m_hNextSkeletonEvent = CreateEvent( NULL, TRUE, FALSE, NULL );
```

The **CreateEvent** function takes four parameters:

- A security descriptor.
- A Boolean that is true if the event is manually reset.
- A Boolean that indicates the initial state of the event.
- A string that contains the name of the event.

All three events have the following characteristics:

1. The security descriptor for the event is NULL.
2. The application resets the event manually.
3. The initial state of each event is not set.
4. The event is unnamed.

Nui_Init next creates the structures and bitmaps that SkeletalViewer requires to render each type of sensor data.

Initialize Skeleton Rendering

The SkeletalViewer sample draws the skeleton by using the Windows GDI subsystem. GDI provides for device-independent output, based on a device context structure. The device context is an opaque

structure that encapsulates device-specific information. GDI supports device contexts for various output targets, such as monitors, printers, and memory. SkeletalViewer draws a skeleton image in a memory region that has the same virtual characteristics as the display device and then passes a pointer to GDI to display the rendered image on the device.

The *CSkeletalViewerApp::Nui_Init* function sets up the required GDI structures, as follows:

```
GetWindowRect(GetDlgItem( m_hWnd, IDC_SKELETALVIEW ), &rc );
int width = rc.right - rc.left;
int height = rc.bottom - rc.top;
HDC hdc = GetDC(GetDlgItem( m_hWnd, IDC_SKELETALVIEW));
m_SkeletonBMP = CreateCompatibleBitmap( hdc, width, height );
m_SkeletonDC = CreateCompatibleDC( hdc );
::ReleaseDC(GetDlgItem(m_hWnd,IDC_SKELETALVIEW), hdc );
m_SkeletonOldObj = SelectObject( m_SkeletonDC, m_SkeletonBMP );
```

GDI setup proceeds as follows:

1. Obtain the size of the window area in which to display the skeleton view by calling the Windows **GetWindowRect** function.
2. Obtain a handle to the device context for the window by calling **GetDC**.
3. Create a bitmap that is compatible with the window device context by calling **CreateCompatibleBitmap**. This bitmap is the same size as the skeletal view area of the window.
4. Create a memory device context that is compatible with the window device context by calling **CreateCompatibleDC**.
5. Release the reference on the window device context by calling **ReleaseDC**.
6. Select the compatible bitmap into the memory device context by calling **SelectObject**. As a result, the bitmap becomes the target for subsequent drawing operations in the memory device context.

The application can write to the memory device context by using the same operations that it would use to write directly to the window. GDI handles all device-specific operations on behalf of the application. When the bitmap is ready for display, SkeletalViewer calls GDI to display it, as described later in this walkthrough.

Initialize Image Rendering

SkeletalViewer uses Direct3D 9 to render depth and video images. By using Direct3D 9, an application can set up multiple back buffers, each of which contains an individual image frame. The application then calls Direct3D 9 to display the frames in sequence by simply swapping buffer pointers, thus performing a bit block transfer (blt). For additional details about swap chains and back buffers, see Flipping Surfaces (Direct3D 9) and What Is a Swap Chain (Direct3D) on the MSDN website.

The *DrawDevice* class, which is defined in DrawDevice.cpp, has methods that create and manipulate Direct3D 9 structures. SkeletalViewer declares m_DrawDepth and m_DrawVideo, which are data members of the *CSkeletalViewerApp* class. They represent *DrawDevice* objects for depth and video images, respectively. *CSkeletalViewerApp::Nui_Init* creates and initializes the Direct3D 9 devices and bitmaps that the application uses with each of these objects.

Nui_Init calls the *DrawDevice::CreateDevice* method on m_DrawDepth to create a Direct3D 9 device for the depth data. It then sets the video parameters for the device by calling *DrawDevice::SetVideoType*. It

passes the width and height of the display window (320x240) and the number of lines to write in a single pass, as follows:

```
hr = m_DrawDepth.CreateDevice( GetDlgItem(m_hWnd, IDC_DEPTHVIEWER ) );
hr = m_DrawDepth.SetVideoType( 320, 240, 320 * 4 );
```

As a result of this call, *SetVideoType* sets the surface format to D3DFMT_X8R8G8B8, which is a 32-bit RGB format with 8 bits for each color channel and a stride of 320x4. *SetVideoType* supports only 32-bit images; other image types are not guaranteed to work properly.

Next, *SetVideoType* creates an additional swap chain. The method sets the parameters for the swap chain in the pp variable and then calls **IDirect3DDevice9::CreateAdditionalSwapChain** to create the chain, as follows:

```
  pp.BackBufferWidth  = m_width;
  pp.BackBufferHeight = m_height;
  pp.Windowed = TRUE;
  pp.SwapEffect = D3DSWAPEFFECT_FLIP;
  pp.hDeviceWindow = m_hwnd;
  pp.BackBufferFormat = D3DFMT_X8R8G8B8;
  pp.Flags =
    D3DPRESENTFLAG_VIDEO | D3DPRESENTFLAG_DEVICECLIP |
    D3DPRESENTFLAG_LOCKABLE_BACKBUFFER;
  pp.PresentationInterval = D3DPRESENT_INTERVAL_IMMEDIATE;
  pp.BackBufferCount = NUM_BACK_BUFFERS;

hr = m_pDevice->CreateAdditionalSwapChain(&pp, &m_pSwapChain);
```

For video data, *Nui_Init* creates a Direct3D device in the same way as for depth data. SkeletalViewer displays video images in a larger window than depth data, so *Nui_Init* calls *SetVideoType* with a width and height of 640x480. The bit format is the same, but the stride is different (640x4), as follows:

```
hr = m_DrawVideo.SetVideoType( 640, 480, 640 * 4 );
```

SetVideoType sets the same surface format and characteristics as for the depth display device.

With the Direct3D setup now complete, *SetVideoType* returns, and *Nui_Init* initializes the Kinect sensor.

Initialize the Kinect Sensor

An application must initialize the Kinect sensor by calling **NuiInitialize** before calling any other **NuiXxx** functions. **NuiInitialize** initializes the internal frame-capture engine, which starts a thread that retrieves data from the Kinect sensor and signals the application when a frame is ready.

After *Nui_Init* sets up the structures that it requires for rendering, it initializes the Kinect sensor as follows:

```
hr = NuiInitialize(
        NUI_INITIALIZE_FLAG_USES_DEPTH_AND_PLAYER_INDEX |
        NUI_INITIALIZE_FLAG_USES_SKELETON |
        NUI_INITIALIZE_FLAG_USES_COLOR);
```

The only parameter to **NuiInitialize** is *dwFlags*, which is a bitwise OR combination of the following NUI_INITIALIZE flags:

NUI_INITIALIZE_FLAG_USES_DEPTH_AND_PLAYER_INDEX
NUI_INITIALIZE_FLAG_USES_COLOR
NUI_INITIALIZE_FLAG_USES_SKELETON
NUI_INITIALIZE_FLAG_USES_DEPTH

The sample uses color, depth, and skeleton tracking, so it specifies the following flags:

NUI_INITIALIZE_FLAG_USES_DEPTH_AND_PLAYER_INDEX
NUI_INITIALIZE_FLAG_USES_SKELETON
NUI_INITIALIZE_FLAG_USES_COLOR

Next, *Nui_Init* enables skeleton tracking by calling **NuiSkeletonTrackingEnable**, as follows:

```
hr = NuiSkeletonTrackingEnable( m_hNextSkeletonEvent, 0 );
```

NuiSkeletonTrackingEnable has the following two parameters:

- A handle to the event to be set when a frame of skeleton data is ready for the application. SkeletalViewer passes *m_hNextSkeletonEvent*, which *CSkeletalViewerApp::Nui_Init* created earlier.
- A set of flags that control skeleton tracking. The beta SDK does not use these flags, so applications should pass zero.

When skeleton tracking is enabled, the runtime processes image and depth data to deliver frames that include skeleton data. You can enable or disable skeleton tracking any time during processing; you are not required to enable it before you open a stream.

Open the Streams

When sensor initialization is complete, *Nui_Init* opens the output streams for the color and depth data by calling **NuiImageStreamOpen**, as follows:

```
hr = NuiImageStreamOpen(
    NUI_IMAGE_TYPE_COLOR,
    NUI_IMAGE_RESOLUTION_640x480,
    0,
    2,
    m_hNextVideoFrameEvent,
    &m_pVideoStreamHandle );

hr = NuiImageStreamOpen(
    NUI_IMAGE_TYPE_DEPTH_AND_PLAYER_INDEX,
    NUI_IMAGE_RESOLUTION_320x240,
    0,
    2,
    m_hNextDepthFrameEvent,
    &m_pDepthStreamHandle );
```

NuiImageStreamOpen has the following parameters:

- The type of image, which is a value of the NUI_IMAGE_TYPE enumeration.
- The requested image resolution, which is a value of the NUI_IMAGE_RESOLUTION enumeration.
- A set of flags that control image preprocessing. The beta SDK ignores these flags, so the sample passes 0.

- The maximum number of lookahead buffers that the application can hold concurrently. This application uses two buffers. By using two buffers, the application can draw with one buffer and the capture driver can fill the other.
- A handle to an event that the runtime sets when the next frame is available.
- A location to receive a handle to the open stream.

The valid image type and resolution depend on the NUI_INITIALIZE flags that the program passed in the *dwFlags* parameter to **NuiInitialize**. The following describes the valid combinations:

- If the application streams color images, *dwFlags* must include NUI_INITIALIZE_FLAG_USES_COLOR.

  Valid resolutions are NUI_IMAGE_RESOLUTION_1280x1024 and NUI_IMAGE_RESOLUTION_640x480.

- If the application streams depth and player index data, *dwFlags* must include NUI_INITIALIZE_FLAG_USES_DEPTH_AND_PLAYER_INDEX.

  Valid resolutions for depth and player index data are NUI_IMAGE_RESOLUTION_320x240 and NUI_IMAGE_RESOLUTION_80x60.

Start the Data Processing Thread

After **NuiImageStreamOpen** returns, *Nui_Init* has the following two additional items of infrastructure to set up:

- An event to signal when the process stops.
- A thread on which to process the data from the Kinect sensor.

Nui_Init calls Windows SDK functions to create both items, as follows:

```
m_hEvNuiProcessStop=CreateEvent(NULL,FALSE,FALSE,NULL);
m_hThNuiProcess=CreateThread(NULL,0,Nui_ProcessThread,this,0,NULL);
```

The **CreateThread** function creates a new thread and immediately begins to execute the *Nui_ProcessThread* method of the *CSkeletalViewerApp* object.

Process Sensor Data

The Kinect sensor supports isochronous data transfer; that is, it supplies frames of data at regular intervals whether or not an application is waiting to receive them.

Sensor data processing in SkeletalViewer is controlled by a loop in the *CSkeletalViewerApp::Nui_ProcessThread* method. The following pseudocode shows the basic flow of the loop:

```
while(1)
{
    // Wait for event.

    // If the stop event occurs, stop looping and exit

    // Calculate frames per second.

    // Blank the skeleton display.
```

```
    // Process frame events
    switch(EventType) {
        case Depth:
            Nui_GotDepthAlert();
            break;
        case Video:
            Nui_GotVideoAlert();
            break;
        case Skeleton:
            Nui_GotSkeletonAlert( );
            break;
    }
}
```

The loop waits for an event to be signaled. If the stop event is signaled, the loop stops processing and the processing thread exits. If the event indicates that a depth frame, a skeleton frame, or a color frame is ready, the loop proceeds as follows:

- Calculates the number of frames per second and displays this number on the screen.
- Clears the skeleton frame display if more than a quarter-second (250 milliseconds) has elapsed since the event indicated a skeleton frame. The NUI API signals a skeleton event when a skeleton appears, but not when a skeleton disappears. Therefore, the application clears the skeleton data after a brief interval.
- Processes the individual frame.

The remainder of this section describes how SkeletalViewer renders each type of data.

Depth and Video Images

The NUI Image Camera API returns depth and video frames in the following format:

```
typedef struct _NUI_IMAGE_FRAME
{
  LARGE_INTEGER             liTimeStamp;
  DWORD                     dwFrameNumber;
  NUI_IMAGE_TYPE            eImageType;
  NUI_IMAGE_RESOLUTION      eResolution;
  NuiImageBuffer *          pFrameTexture;
  DWORD                     dwFrameFlags_NotUsed;
  NUI_IMAGE_VIEW_AREA       ViewArea_NotUsed;
} NUI_IMAGE_FRAME;
```

Currently, the frame contains the following information:

- The time stamp of the frame, in milliseconds, which indicates the elapsed time since **NuiInitialize** was called.
- A frame number. Frame numbers are sequential starting when the Kinect sensor is initialized, and they reset to zero when the device is unplugged or uninitialized.
- The type of image. The following enumeration values are currently supported:

  NUI_IMAGE_TYPE_DEPTH_AND_PLAYER_INDEX
  NUI_IMAGE_TYPE_COLOR
  NUI_IMAGE_TYPE_COLOR_YUV

NUI_IMAGE_TYPE_COLOR_RAW_YUV
NUI_IMAGE_TYPE_DEPTH

- The resolution of the image, which is indicated by one of the following enumeration values:

  NUI_IMAGE_RESOLUTION_80x60
  NUI_IMAGE_RESOLUTION_320x240
  NUI_IMAGE_RESOLUTION_640x480
  NUI_IMAGE_RESOLUTION_1280x1024

- A pointer to a **NuiImageBuffer** object.

The remaining two members are currently not used.

Depth Frames

When a depth frame event occurs, the *Nui_ProcessThread* method calls *CSkeletalViewerApp::Nui_GotDepthAlert* to process the depth data. *Nui_GotDepthAlert* retrieves the ready frame from the capture engine and renders it on the screen.

To retrieve the frame, *Nui_GotDepthAlert* calls **NuiImageStreamGetNextFrame**, which takes as parameters the handle to the depth frame, the number of milliseconds to wait for a frame, and a pointer to a buffer to receive the frame, as follows:

```
const NUI_IMAGE_FRAME * pImageFrame = NULL;

HRESULT hr = NuiImageStreamGetNextFrame(
    m_pDepthStreamHandle,
    0,
    &pImageFrame );
NuiImageBuffer * pTexture = pImageFrame->pFrameTexture;
```

The returned pImageFrame parameter points to a NUI_IMAGE_FRAME structure. The depth data is available through the **pFrameTexture** member of the structure. To simplify references, the code assigns the pointer to a local variable that is named pTexture.

Next, *Nui_GotDepthAlert* locks the returned frame to prevent the underlying data from changing while the application is processing it. To lock the frame, *Nui_GotDepthAlert* calls the **LockRect** method on the **NuiImageBuffer** object at pTexture, as follows:

```
KINECT_LOCKED_RECT LockedRect;
pTexture->LockRect( 0, &LockedRect, NULL, 0 );
```

The parameters to **LockRect** are a level, the address of a KINECT_LOCKED_RECT structure, a pointer to a rectangle that describes the region to lock, and a set of flags. Currently, **LockRect** ignores all parameters except the structure pointer, and it always locks the entire frame. The KINECT_LOCKED_RECT structure has two members:

- **Pitch**, an integer that indicates the number of bytes in one row of the image.
- **pBits**, a pointer to the locked bits in the image.

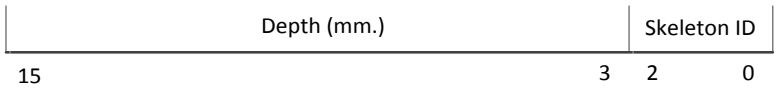
Note Currently, the frame is simply a memory buffer, rather than a Direct3D texture, so locking it is not strictly required. However, the NUI API provides the **LockRect** method to ensure compatibility with possible future implementations, and applications should use this method.

The **LockRect** method returns a pointer to a locked region. *Nui_GotDepthAlert* then loops through the region, unpacking the data, assigning RGB values for display, and writing the RGB values to a linear array that is named *m_rgbWk*, as follows:

```
BYTE * pBuffer = (BYTE*) LockedRect.pBits;

// draw the bits to the bitmap
RGBQUAD * rgbrun = m_rgbWk;
USHORT * pBufferRun = (USHORT*) pBuffer;
for( int y = 0 ; y < 240 ; y++ ) {
    for( int x = 0 ; x < 320 ; x++ ){
        RGBQUAD quad = Nui_ShortToQuad_Depth( *pBufferRun );
        pBufferRun++;
        *rgbrun = quad;
        rgbrun++;
    }
}
```

The internal *Nui_ShortToQuad_Depth* function transforms each 16-bit depth value into a format that is appropriate for display. Because the application specified NUI_IMAGE_TYPE_DEPTH_AND_PLAYER_INDEX in the call to **NuiInitialize**, the 16-bit depth value is laid out as follows:



Nui_ShortToQuad_Depth shifts the bits to remove the skeleton ID and assigns the result to *RealDepth*. It also stores the player ID in the *Player* variable so that it can display each person in the image in a different color. Then it inverts the remaining bits to generate an 8-bit intensity value that is appropriate for display, so that objects that are closer to the camera appear brighter and objects that are farther away appear darker. It ignores the most significant bit in this calculation. The following code example shows this transformation:

```
USHORT RealDepth = (s & 0xfff8) >> 3;
USHORT Player = s & 7;
BYTE l = 255 - (BYTE)(256*RealDepth/0x0fff);
```

It then converts the byte value to the **rgbRed**, **rgbBlue**, and **rgbGreen** members of an RGBQUAD structure for display with color coding to indicate the depth of each visible skeleton in the image, as follows:

```
RGBQUAD q;
q.rgbRed = q.rgbBlue = q.rgbGreen = 0;
switch( Player )
{
    case 0:
        q.rgbRed = l / 2;
        q.rgbBlue = l / 2;
        q.rgbGreen = l / 2;
        break;
    case 1:
        q.rgbRed = l;
        break;
```

```
    case 2:
        q.rgbGreen = l;
        break;
    case 3:
        q.rgbRed = l / 4;
        q.rgbGreen = l;
        q.rgbBlue = l;
        break;
    case 4:
        q.rgbRed = l;
        q.rgbGreen = l;
        q.rgbBlue = l / 4;
        break;
    case 5:
        q.rgbRed = l;
        q.rgbGreen = l / 4;
        q.rgbBlue = l;
        break;
    case 6:
        q.rgbRed = l / 2;
        q.rgbGreen = l / 2;
        q.rgbBlue = l;
        break;
    case 7:
        q.rgbRed = 255 - ( l / 2 );
        q.rgbGreen = 255 - ( l / 2 );
        q.rgbBlue = 255 - ( l / 2 );
}
```

Nui_ShortToQuad_Depth returns the RGB_QUAD structure, and *Nui_GotDepthAlert* assigns it to the next sequential position in the m_rgbWk array. The RGBQUAD type is defined in wingdi.h.

Nui_GotDepthAlert then displays the frame by calling the internal *DrawFrame* method on the m_DrawDepth DrawDevice object, as follows:

```
    m_DrawDepth.DrawFrame( (BYTE*) m_rgbWk );
```

For details about *DrawFrame*, see the following section, "Depth and Video Frame Display."

After *DrawFrame* returns, *Nui_GotDepthAlert* completes its processing by releasing the depth frame, as follows:

```
NuiImageStreamReleaseFrame( m_pDepthStreamHandle, pImageFrame );
```

Depth and Video Frame Display

SkeletalViewer displays depth and video frames by using a *DrawDevice* object. The *DrawDevice* class is defined and implemented in DrawDevice.h and DrawDevice.cpp.

When SkeletalViewer has calculated the pixel values to represent a frame of data, it calls *DrawFrame* on the *DrawDevice* object for that image type, passing as a parameter pBits, which is a pointer to a byte-array that contains the data. *DrawFrame* copies the data into the Direct3D buffers that it uses for display and then presents the filled buffer to Direct3D to display in the window.

DrawFrame starts by setting up pointers and checking the required infrastructure for frame display, as follows:

```
CComPtr< IDirect3DSurface9 > pSurf = NULL;
CComPtr< IDirect3DSurface9 > pBB = NULL;

if (m_pDevice == NULL || m_pSwapChain == NULL) {
    return S_OK;
}

hr = m_pSwapChain->GetBackBuffer(0, D3DBACKBUFFER_TYPE_MONO, &pSurf);
if (FAILED(hr)) { goto done; }
```

This code example does the following:

- Uses **CComPtr** to access the **IDirect3DSurface9** interfaces for the surface and back buffers through pSurf and pBB, respectively.
- Checks to ensure that the Direct3D device and the swap chain exist. By default, Direct3D 9 creates a single swap chain for each device. The SkeletalViewer creates an additional swap chain so that it can render images in the background and then swap buffers for faster display.
- Gets a pointer to the **IDirect3DSurface9** interface for the surface buffer from the swap chain.

The *lr* variable is a rectangle that contains the output buffer. *DrawFrame* locks the output buffer and copies the data into it, as follows:

```
hr = pSurf->LockRect(&lr, NULL, D3DLOCK_NOSYSLOCK );

BYTE * pDst = (BYTE*) lr.pBits;
BYTE * pSrc = pBits;

for( int y = 0 ; y < (int) m_height ; y++ )
{
    memcpy( pDst, pSrc, m_lDefaultStride );
    pDst += lr.Pitch;
    pSrc += m_lDefaultStride;
}

hr = pSurf->UnlockRect();
```

The source frame is *pBits* and the destination buffer is *lr.pBits*, which is the locked rectangle on the surface buffer. *DrawFrame* copies one line at a time from the source frame into the destination. The line length depends on the image type and was set up previously by the *DrawDevice::SetVideoType* method. When the copy operation is complete, *DrawFrame* unlocks the surface buffer. The surface buffer surface now contains the image.

Next, *DrawFrame* retrieves a pointer to the device back buffer and fills the buffer with a dark blue color. It then copies the contents of the surface buffer from the swap chain into the device back buffer by using a linear texture filter. Finally, it calls the **IDirect3DDevice9::Present** method of the device back buffer to display the generated image, as follows:

```
hr = m_pDevice->GetBackBuffer(0, 0, D3DBACKBUFFER_TYPE_MONO, &pBB);
hr = m_pDevice->ColorFill(pBB, NULL, D3DCOLOR_XRGB(0, 0, 0x80));
hr = m_pDevice->StretchRect(pSurf, NULL, pBB, &m_rcDest,
      D3DTEXF_LINEAR);
```

```
hr = m_pDevice->Present(NULL, NULL, NULL, NULL);
```

Skeleton Images

SkeletalViewer renders skeleton data as line and point drawings, rather than assembling a frame and then performing a bit blt. To draw the skeletons, the sample uses the Windows GDI subsystem. GDI provides for device-independent output, so that an application can call the same methods to render to various display devices without concern for the characteristics of the underlying device.

Skeleton Frame

A frame of skeleton data has the following format:

```
typedef struct _NUI_SKELETON_FRAME
{
  LARGE_INTEGER         liTimeStamp;
  DWORD                 dwFrameNumber;
  DWORD                 dwFlags;
  Vector4               vFloorClipPlane;
  Vector4               vNormalToGravity;
  NUI_SKELETON_DATA     SkeletonData[NUI_SKELETON_COUNT];
} NUI_SKELETON_FRAME;
```

When skeleton tracking is enabled, the internal pipeline in the beta SDK runtime processes depth frame data to create skeleton frames. The **dwFrameNumber** member of the NUI_SKELETON_FRAME structure contains the frame number of the depth image that was processed to create the skeleton frame. The API signals a skeleton event every time it processes a depth frame, so that applications have access to the floor clip plane data, even if no skeleton is present.

The beta SDK supports active tracking of up to two skeletons and passive tracking of up to four more. The skeleton frame returns data for all skeletons—whether actively or passively tracked—in the **SkeletonData** member. **SkeletonData** is an array of NUI_SKELETON_DATA structures in the following format:

```
typedef struct _NUI_SKELETON_DATA
{
  NUI_SKELETON_TRACKING_STATE eTrackingState;
  DWORD dwTrackingID;
  DWORD dwEnrollmentIndex;
  DWORD dwUserIndex;
  Vector4 Position;
  Vector4 SkeletonPositions[NUI_SKELETON_POSITION_COUNT];
  NUI_SKELETON_POSITION_TRACKING_STATE
     eSkeletonPositionTrackingState[NUI_SKELETON_POSITION_COUNT];
  DWORD dwQualityFlags;
} NUI_SKELETON_DATA;
```

The **eTrackingState** member indicates whether the structure contains tracking data for all points on the skeleton or just overall data:

- For passively tracked skeletons, this member is equal to NUI_SKELETON_POSITION_ONLY. The **Position** member contains valid data, but the **SkeletonPositions**, **eSkeletonPositionTrackingState**, and **dwQualityFlags** members do not. The **Position** member describes the center of mass for the skeleton.

- For actively tracked skeletons, the **eTrackingState** member is equal to NUI_SKELETON_TRACKED, the **Position** member indicates the center of mass for the skeleton, and the **SkeletonPositions** array contains a skeleton joint position in each element. The **eSkeletonPositionTrackingState** array contains an enumerator that indicates whether the corresponding position is being tracked or inferred. The **dwQualityFlags** member indicates whether any portion of the skeleton is out of the frame.
- If the number of tracked skeletons is less than NUI_SKELETON_COUNT, the **eTrackingState** member is set to NUI_SKELETON_NOT_TRACKED for each array element that does not contain tracked skeleton data. The array is not guaranteed to have tracked skeleton data starting from element 0; that is, there can be gaps in the array.

Skeleton tracking IDs range from 1 to 6. A tracking ID equal to 0 indicates that the skeleton is not currently being tracked. The tracking ID always appears at the same index in the **SkeletonData** array. The enrollment index is not used in the current beta SDK, and the **dwUserIndex** member is always set to XUSER_INDEX_NONE.

The skeleton data is returned as a set of 20 points, one for each of the skeleton positions that are defined by the NUI_SKELETON_POSITION_INDEX enumeration.

Skeleton Drawing

When a skeleton event occurs, the *Nui_ProcessThread* method calls *Nui_GotSkeletonAlert*. *Nui_GotSkeletonAlert* proceeds as follows:

1. Retrieves a frame of skeleton data by calling **NuiSkeletonGetNextFrame**.
2. Checks the frame to determine whether it describes a skeleton and, if not, returns.
3. Smooths the frame for display by calling **NuiTransformSmooth**.
4. Draws each of the currently tracked skeletons by calling *Nui_DrawSkeleton*.
5. Displays the rendered image on the screen by calling *Nui_DoDoubleBuffer*.

NuiSkeletonGetNextFrame takes the following two parameters:

- A DWORD that indicates the number of milliseconds to wait for the frame.
- A pointer to a location in which to return a NUI_SKELETON_FRAME structure.

The following example shows this:

```
HRESULT hr = NuiSkeletonGetNextFrame( 0, &SkeletonFrame );
```

The skeleton runtime signals a skeleton event each time it processes a depth frame, so it is possible to retrieve a skeleton frame that does not contain skeleton data. If the current frame does not contain a skeleton, *Nui_GotSkeletonAlert* simply returns, as the following code example shows:

```
bool bFoundSkeleton = true;
for( int i = 0 ; i < NUI_SKELETON_COUNT ; i++ )
{
  if( SkeletonFrame.SkeletonData[i].eTrackingState == NUI_SKELETON_TRACKED )
  {
    bFoundSkeleton = false;
  }
}
if( bFoundSkeleton )
```

```
{
  return;
}
```

Sequential display of skeleton frames can result in jittery images. To reduce the jitter, *Nui_GotSkeletonAlert* calls **NuiTransformSmooth** to apply a smoothing filter to the skeleton data, as follows:

```
NuiTransformSmooth(&SkeletonFrame, NULL);
```

NuiTransformSmooth filters the data for all the tracked skeletons.

Next, *Nui_GotSkeletonAlert* restarts the skeleton timer for screen blanking and calls *Nui_DrawSkeleton* for each tracked skeleton, as follows:

```
m_bScreenBlanked = false;
m_LastSkeletonFoundTime = -1;
bool bBlank = true;
if( SkeletonFrame.SkeletonData[i].eTrackingState == NUI_SKELETON_TRACKED )
{
  Nui_DrawSkeleton( bBlank, &SkeletonFrame.SkeletonData[i],
      GetDlgItem( m_hWnd, IDC_SKELETALVIEW ), i );
  bBlank = false;
}
```

Nui_DrawSkeleton takes the following four parameters:

- A Boolean that indicates whether to clear the window before drawing the skeleton. This parameter is set to true for the first skeleton in the frame and false for additional skeletons.
- A pointer to the data for this skeleton.
- A handle to the window area in which to draw the skeleton.
- An integer that indicates the pen color. The pen color is based on the index of the skeleton in the **SkeletonData** array.

The first time *Nui_DrawSkeleton* is called, it creates pens for drawing the skeletons by calling the Windows GDI **CreatePen** function, as follows:

```
m_Pen[0] = CreatePen( PS_SOLID, width / 80, RGB(0, 255, 0) );
m_Pen[1] = CreatePen( PS_SOLID, width / 80, RGB( 0, 0, 255 ) );
m_Pen[2] = CreatePen( PS_SOLID, width / 80, RGB( 255, 0, 0 ) );
m_Pen[3] = CreatePen( PS_SOLID, width / 80, RGB(255, 255, 64 ) );
m_Pen[4] = CreatePen( PS_SOLID, width / 80, RGB( 255, 64, 255 ) );
m_Pen[5] = CreatePen( PS_SOLID, width / 80, RGB( 128, 128, 255 ) );
```

Each pen draws a solid line that is the width of the window area divided by 80, which makes the width of the skeleton proportional to the total area that is available for display.

Nui_DrawSkeleton then clears the window if the caller passed true for the first parameter and retrieves the coordinates of the client window so that it can scale the image to the window size, as follows:

```
if( bBlank )
{
  PatBlt( m_SkeletonDC, 0, 0, width, height, BLACKNESS );
}
```

```
int scaleX = width;
int scaleY = height;
float fx=0,fy=0;
```

The next step in drawing the skeleton is to map the skeleton coordinates into depth image coordinates, as follows:

```
int i;
for (i = 0; i < NUI_SKELETON_POSITION_COUNT; i++)
{
  NuiTransformSkeletonToDepthImageF( pSkel->SkeletonPositions[i], &fx, &fy );
  m_Points[i].x = (int) ( fx * scaleX + 0.5f );
  m_Points[i].y = (int) ( fy * scaleY + 0.5f );
}
```

NuiTransformSkeletonToDepthImageF returns floating-point x and y coordinates in the depth image that correspond to the skeleton position. The transformation is required because the skeleton data and the depth data are based on different coordinate systems. The center of the depth sensor is at (0, 0, 0) in skeleton space and at (160, 120) in depth image coordinates. *Nui_DrawSkeleton* then calculates integer values for the coordinates that it can use later to draw the skeleton.

When the data transformation is complete, *Nui_DrawSkeleton* selects the pen to use for this skeleton and begins to draw. By using the enumeration values that identify the skeleton positions, rendering code can group the following related body parts:

- The center of the hip, the spine, the center of the shoulder, and the head.
- The center of the shoulder, the left shoulder, the left elbow, the left wrist, and the left hand.
- The center of the shoulder, the right shoulder, the right elbow, the right wrist, and the right hand.
- The center of the hip, the left hip, the left knee, the left ankle, and the left foot.
- The center of the hip, the right hip, the right knee, the right ankle, and the right foot.

The *CSkeletalViewerApp::Nui_DrawSkeletonSegment* method draws a segment of the skeleton. *Nui_DrawSkeleton* calls this method once for each group of body parts in the preceding list. For each group of parts, *Nui_DrawSkeletonSegment* receives a frame of smoothed skeleton data, the number of joints in this segment of the skeleton, and a list of the joints as a variable argument. The following shows the code for this method:

```
void CSkeletalViewerApp::Nui_DrawSkeletonSegment(
    NUI_SKELETON_DATA * pSkel, int numJoints, ... )
{
  va_list vl;
  va_start(vl,numJoints);
  POINT segmentPositions[NUI_SKELETON_POSITION_COUNT];

  for (int iJoint = 0; iJoint < numJoints; iJoint++)
  {
    NUI_SKELETON_POSITION_INDEX jointIndex =
       va_arg(vl,NUI_SKELETON_POSITION_INDEX);
    segmentPositions[iJoint].x = m_Points[jointIndex].x;
    segmentPositions[iJoint].y = m_Points[jointIndex].y;
  }

  Polyline(m_SkeletonDC, segmentPositions, numJoints);
```

```
  va_end(vl);
}
```

The method proceeds by stepping through the list of joints and creating a line that connects them. It then calls the Windows GDI **Polyline** function to draw the line.

Finally, *Nui_DrawSkeleton* draws the joints in a different color from the lines. For each joint, it creates a different pen color, selects the pen, moves to the joint position, and draws the joint, as follows:

```
for (i = 0; i < NUI_SKELETON_POSITION_COUNT ; i++)
{
    HPEN hJointPen;

    hJointPen=CreatePen(PS_SOLID,9, g_JointColorTable[i]);
    hOldObj=SelectObject(m_SkeletonDC,hJointPen);

    MoveToEx( m_SkeletonDC, m_Points[i].x, m_Points[i].y, NULL );
    LineTo( m_SkeletonDC, m_Points[i].x, m_Points[i].y );

    SelectObject( m_SkeletonDC, hOldObj );
    DeleteObject(hJointPen);
}
```

The m_SkeletonDC buffer now contains a line drawing of the first skeleton. *Nui_DrawSkeleton* exits, which returns control to *CSkeletalViewerApp::Nui_GotSkeletonAlert*. That method, in turn, continues to loop through the skeleton data until all tracked skeletons have been drawn in the buffer.

Skeleton Display

When the buffer contains all the tracked skeletons, *CSkeletalViewerApp::Nui_GotSkeletonAlert* calls *CSkeletalViewerApp::Nui_DoDoubleBuffer* to present the buffer on the display. *Nui_DoDoubleBuffer* takes two parameters: a handle to the region in which to display the bitmap and a handle to the device context. It calls the Windows **GetDC** function to get a handle to the device context for the display window, calls the Windows **BitBlt** function to transfer the buffer, and then releases the display device context, as follows:

```
void CSkeletalViewerApp::Nui_DoDoubleBuffer(HWND hWnd, HDC hDC)
{
    RECT rct;
    GetClientRect(hWnd, &rct);
    int width = rct.right;
    int height = rct.bottom;

    HDC hdc = GetDC( hWnd );

    BitBlt( hdc, 0, 0, width, height, hDC, 0, 0, SRCCOPY );

    ReleaseDC( hWnd, hdc );

}
```

Exit from the Application

When the user closes the window, the *WndProc* callback function gets the WM_CLOSE message. In response, it calls the Windows **DestroyWindow** function. **DestroyWindow**, in turn, sends a WM_DESTROY message to *WndProc*. In response to this message, the *WndProc* function calls *Nui_UnInit*, which performs the cleanup that is required before the program exits.

Nui_UnInit first deletes the device context and bitmap that *Nui_Initialize* created for skeleton processing and the pens that *Nui_DrawSkeleton* created, as follows:

```
::SelectObject( m_SkeletonDC, m_SkeletonOldObj );
DeleteDC( m_SkeletonDC );
DeleteObject( m_SkeletonBMP );
if( m_Pen[0] != NULL )
{
    DeleteObject(m_Pen[0]);
    DeleteObject(m_Pen[1]);
    DeleteObject(m_Pen[2]);
    DeleteObject(m_Pen[3]);
    DeleteObject(m_Pen[4]);
    DeleteObject(m_Pen[5]);
    ZeroMemory(m_Pen,sizeof(m_Pen));
}
```

Nui_UnInit then stops the sensor data processing thread, closes the handle to the stop-processing event, and calls **NuiShutdown**, as follows:

```
    if(m_hEvNuiProcessStop!=NULL)
    {
        // Signal the thread
        SetEvent(m_hEvNuiProcessStop);

        // Wait for thread to stop
        if(m_hThNuiProcess!=NULL)
        {
            WaitForSingleObject(m_hThNuiProcess,INFINITE);
            CloseHandle(m_hThNuiProcess);
        }
        CloseHandle(m_hEvNuiProcessStop);
    }
NuiShutdown( );
```

NuiShutdown stops frame capture and frees internal data structures for the runtime.

Nui_Uninit then closes the remaining event handles, deletes the Direct3D 9 devices that it created to render depth and video data, and returns, as follows:

```
if( m_hNextSkeletonEvent
       && ( m_hNextSkeletonEvent != INVALID_HANDLE_VALUE ) )
{
    CloseHandle( m_hNextSkeletonEvent );
    m_hNextSkeletonEvent = NULL;
```

```
}
if( m_hNextDepthFrameEvent
    && ( m_hNextDepthFrameEvent != INVALID_HANDLE_VALUE ) )
{
    CloseHandle( m_hNextDepthFrameEvent );
    m_hNextDepthFrameEvent = NULL;
}
if( m_hNextVideoFrameEvent
    && ( m_hNextVideoOFrameEvent != INVALID_HANDLE_VALUE ) )
{
    CloseHandle( m_hNextVideoFrameEvent );
    m_hNextVideoFrameEvent = NULL;
}
m_DrawDepth.DestroyDevice( );
m_DrawVideo.DestroyDevice( );
```

Finally, the *WndProc* callback function deletes the font object that it created to display the frames per second and calls the Windows **PostQuitMessage** function to inform the system that the thread is terminating.

PART 3—C# SkeletalViewer Sample

The C# SkeletalViewer sample uses the managed NUI API to capture depth data, color, and skeletal tracking data, and then renders the images on the screen by using WPF. This walkthrough assumes that you are familiar with WPF and its component namespaces. For an introduction to WPF, see "Resources" in Part 4 of this document.

Program Basics

The C# SkeletalViewer application is installed with the Kinect for Windows Software Development Kit (SDK) Beta samples in %KINECTSDK_DIR%\Samples\KinectSDKSamples.zip

The sample is implemented in the following files:

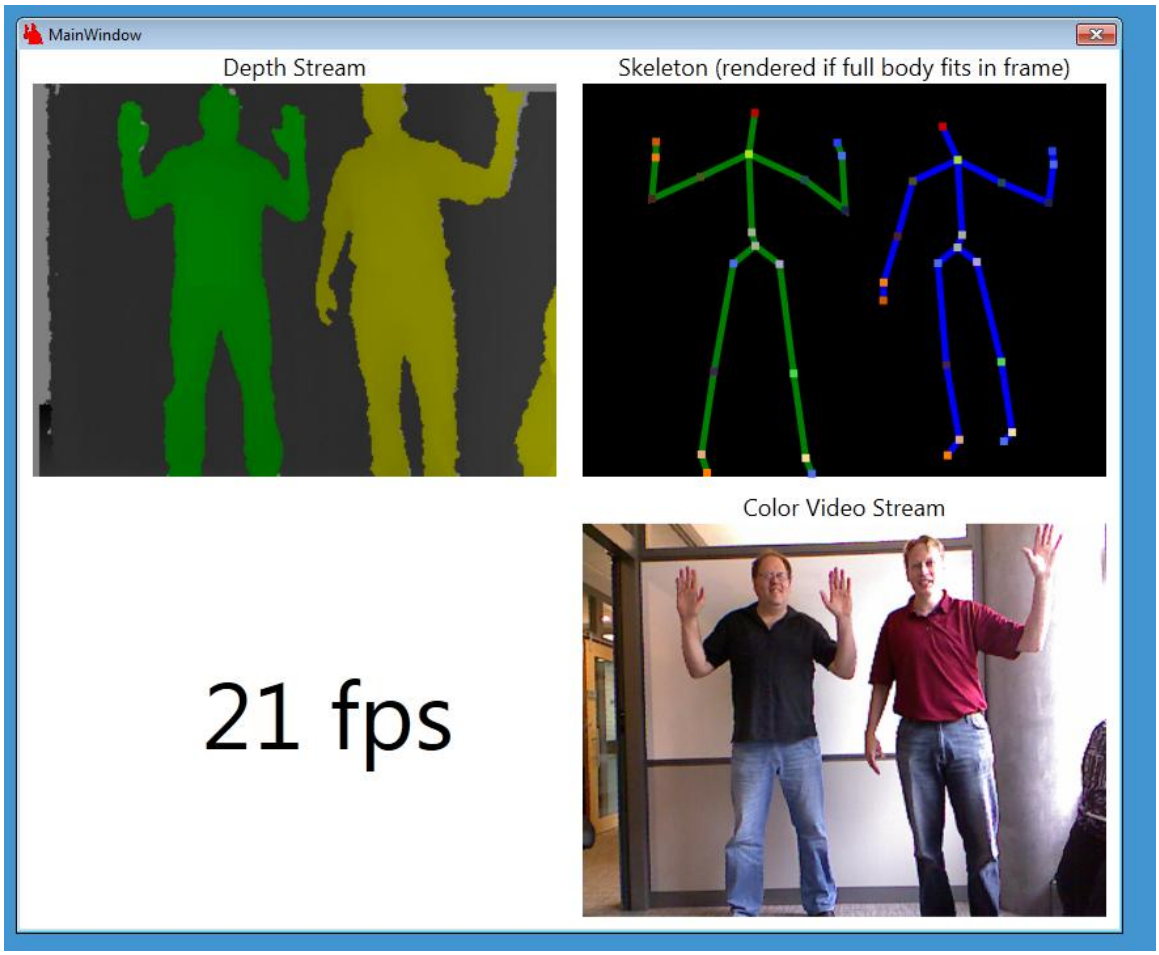
- App.xaml declares application-level resources.
- App.xaml.cs contains the code behind app.xaml.
- MainWindow.xaml declares the layout of the main application window.
- MainWindow.xaml.cs contains the code behind the main window, which implements NUI API initialization, processing and display.
- SkeletalViewer.ico defines the application icon that is used in the title bar and the task bar.

To build and run the sample

1. In Windows Explorer, navigate to the SkeletalViewer\CS directory.
2. Double-click the icon for the .sln (solution) file to open the file in Visual Studio.
3. Build the application.
4. Press CTRL+F5 to run the sample.

The solution file for the sample targets the x86 platform, because this beta SDK includes only x86 libraries.

The following illustration shows the main application window for the C# SkeletalViewer.



This document walks you through the sample and focuses on the Kinect-specific aspects of the code for data capture.

Note This document includes excerpts from the sample program, most of which have been edited for brevity and readability. In particular, most routine error correction code has been removed. For the complete code, see the SkeletalViewer sample in the Nui\SkeletalViewer\CS folder.

Create the Main Window

The C# SkeletalViewer sample uses WPF to create a window that displays color video, depth, and skeleton frames, along with an approximation of the number of frames per second. It uses the **System.Windows** namespace for basic Window element classes and uses several additional subordinate **System.Windows** namespaces for specific features, such as imaging. Most importantly, it includes the **Microsoft.Research.Kinect.Nui** namespace for access to the managed code interface for the beta SDK.

The *SkeletalViewer* namespace declares a public **Window** class object that is named *MainWindow*, which contains the following:

- Code to initialize global variables.
- The *Window_Loaded* method, which handles the **Window.Loaded** event and initializes the NUI API runtime, opens streams, and connects events and their event handlers.
- The *nui_DepthFrameReady*, *nui_SkeletonFrameReady*, and *nui_ColorFrameReady* event handlers.
- The *convertDepthFrame* method, which converts 16-bit depth data to a 32-bit format for rendering.
- The *getBodySegment* method, which draws a part of the skeleton.
- The *Window_Closed* method, which handles the **Window.Closed** event.

The *MainWindow* class initializes the window and then creates and initializes global variables as follows:

- Declares *nui* as a **Runtime** object, which represents the Kinect sensor instance.
- Initializes several variables—*totalFrames*, *lastFrames*, and *lastTime*—that are used to calculate the number of frames per second.
- Defines *depthFrame32* as a byte array that is large enough to hold an entire frame of depth data at 32 bits per pixel and defines the RED_IDX, GREEN_IDX, and BLUE_IDX constants for use in indexing that array.
- Creates *jointColors* as a **System.Collections.Generic.Dictionary** class object that associates a brush color with each skeleton joint for later use in rendering.

The following shows the initialization code from *MainWindow*:

```
namespace SkeletalViewer
public partial class MainWindow : Window
{
  public MainWindow()
  {
    InitializeComponent();
  }

  Runtime nui;
  int totalFrames = 0;
  int lastFrames = 0;
  DateTime lastTime = DateTime.MaxValue;

  const int RED_IDX = 2;
  const int GREEN_IDX = 1;
  const int BLUE_IDX = 0;
  byte[] depthFrame32 = new byte[320 * 240 * 4];

  Dictionary<JointID,Brush> jointColors =
    new Dictionary<JointID,Brush>() {
    {JointID.HipCenter, new SolidColorBrush(Color.FromRgb(169, 176, 155))},
    {JointID.Spine, new SolidColorBrush(Color.FromRgb(169, 176, 155))},
    {JointID.ShoulderCenter, new SolidColorBrush(Color.FromRgb(168, 230,29))},
    {JointID.Head, new SolidColorBrush(Color.FromRgb(200, 0, 0))},
```

```
    {JointID.ShoulderLeft, new SolidColorBrush(Color.FromRgb(79, 84, 33))},
    {JointID.ElbowLeft, new SolidColorBrush(Color.FromRgb(84, 33, 42))},
    {JointID.WristLeft, new SolidColorBrush(Color.FromRgb(255, 126, 0))},
    {JointID.HandLeft, new SolidColorBrush(Color.FromRgb(215, 86, 0))},
    {JointID.ShoulderRight, new SolidColorBrush(Color.FromRgb(33, 79, 84))},
    {JointID.ElbowRight, new SolidColorBrush(Color.FromRgb(33, 33, 84))},
    {JointID.WristRight, new SolidColorBrush(Color.FromRgb(77, 109, 243))},
    {JointID.HandRight, new SolidColorBrush(Color.FromRgb(37, 69, 243))},
    {JointID.HipLeft, new SolidColorBrush(Color.FromRgb(77, 109, 243))},
    {JointID.KneeLeft, new SolidColorBrush(Color.FromRgb(69, 33, 84))},
    {JointID.AnkleLeft, new SolidColorBrush(Color.FromRgb(229, 170, 122))},
    {JointID.FootLeft, new SolidColorBrush(Color.FromRgb(255, 126, 0))},
    {JointID.HipRight, new SolidColorBrush(Color.FromRgb(181, 165, 213))},
    {JointID.KneeRight, new SolidColorBrush(Color.FromRgb(71, 222, 76))},
    {JointID.AnkleRight, new SolidColorBrush(Color.FromRgb(245, 228, 156))},
    {JointID.FootRight, new SolidColorBrush(Color.FromRgb(77,  109, 243))}
  };
// . . . SkeletalViewer namespace code continues
```

Initialize Runtime

The *Window_Loaded* method creates the *nui* runtime object, opens the video and depth streams, and sets up the event handlers that the runtime calls when a video, depth, or skeleton frame is ready, as follows:

```
private void Window_Loaded(object sender, EventArgs e)
{
  nui = new Runtime();
  try
  {
    nui.Initialize(RuntimeOptions.UseDepthAndPlayerIndex |
          RuntimeOptions.UseSkeletalTracking | RuntimeOptions.UseColor);
  }
  catch (InvalidOperationException)
  {
    // Display error message; omitted for space
    return;
  }

  try
  {
    nui.VideoStream.Open(ImageStreamType.Video, 2,
        ImageResolution.Resolution640x480, ImageType.Color);
    nui.DepthStream.Open(ImageStreamType.Depth, 2,
        ImageResolution.Resolution320x240, ImageType.DepthAndPlayerIndex);
  }
  catch (InvalidOperationException)
  {
    // Display error message; omitted for space
    return;
  }
  lastTime = DateTime.Now;
```

```
  nui.DepthFrameReady += new EventHandler<ImageFrameReadyEventArgs>
        (nui_DepthFrameReady);
  nui.SkeletonFrameReady +=
        new EventHandler<SkeletonFrameReadyEventArgs>
        (nui_SkeletonFrameReady);
  nui.VideoFrameReady +=
        new EventHandler<ImageFrameReadyEventArgs>
        (nui_ColorFrameReady);
}
```

An application must initialize the Kinect sensor by calling **Runtime.Initialize** before calling any other methods on the Runtime object. **Runtime.Initialize** initializes the internal frame-capture engine, which starts a thread that retrieves data from the Kinect sensor and signals the application when a frame is ready. It also initializes the subsystems that collect and process the sensor data. The Initialize method throws **InvalidOperationException** if it fails to find a Kinect sensor, so the call to **Runtime.Initialize** appears in a **try/catch** block.

The only parameter to **Runtime.Initialize** is *options*, which is a bitwise OR combination of the following values from the **RuntimeOptions** enumeration:

- **UseDepthAndPlayerIndex**
- **UseColor**
- **UseSkeletalTracking**
- **UseDepth**

The sample uses color, depth, player index, and skeleton tracking, so it specifies **UseDepthAndPlayerIndex, UseColor,** and **UseSkeletalTracking**.

Next, the *Window_Loaded* method opens the video and depth streams by calling *nui.VideoStream.Open* and *nui.DepthStream.Open*. **VideoStream** and **DepthStream** are properties of the **Runtime** object. The value of each property is an **ImageStream** object that represents the stream. The **ImageStream**.**Open** method has the following four parameters:

- The stream type, which is either **ImageStreamType.Video** or **ImageStreamType.Depth**.
- The number of lookahead buffers that the application maintains. By using two buffers, the application can draw with one buffer and the capture driver can fill the other.
- The image resolution, which is a value of the **ImageResolution** enumeration.
- The image type, which is a value of the **ImageType** enumeration.

The valid image resolution and image type depend on which subsystems are active and are therefore constrained by the *options* that the application specified with **Runtime.Initialize**. If the application specifies a resolution or image type that is not compatible with the initialization options, **ImageStream.Open** method throws an exception.

To stream color images:

- The *options* must include **UseColor**.
- Valid image resolutions are **Resolution1280x1024** and **Resolution640x480**.
- Valid image types are **Color**, **ColorYUV**, and **ColorYUVRaw**.

To stream depth and player index data:

- The *options* must include **UseDepthAndPlayerIndex**.
- Valid resolutions for depth and player index data are **Resolution320x240** and **Resolution80x60**.
- The only valid image type is **DepthAndPlayerIndex**.

Next, *Window_Loaded* saves the current time in *lastTime* to use to calculate frames per second.

Applications can use either a polling model or an event-driven model to retrieve frames. An application that uses the polling model calls **ImageStream.GetNextFrame** or **SkeletonEngine.GetNextFrame** and waits until a frame is ready. Applications that use the event-driven model subscribe to one or more frame-ready events and implement event handlers. The SkeletalViewer sample uses the event-driven model.

To implement the event-driven model, the *Window_Loaded* method creates **DepthFrameReady**, **SkeletonFrameReady**, and **VideoFrameReady** events for the *nui* **Runtime** object and associates them with the event handlers that are named *nui_DepthFrameReady*, *nui_SkeletonFrameReady*, and *nui_ColorFrameReady*, respectively.

When a frame from the Kinect sensor is ready for the application, the runtime signals the relevant event and the associated event handler method runs.

Process Video Data

When a video frame is ready, the runtime signals the **VideoFrameReady** event and calls *nui_ColorFrameReady*. The following shows the **VideoFrameReady** event handling code:

```
void nui_ColorFrameReady(object sender, ImageFrameReadyEventArgs e)
{
  PlanarImage Image = e.ImageFrame.Image;
  video.Source = BitmapSource.Create(
      Image.Width, Image.Height, 96, 96, PixelFormats.Bgr32, null,
      Image.Bits, Image.Width * Image.BytesPerPixel);
}
```

The *e* parameter to this event handler is an object of the **ImageFrameReadyEventArgs** class, which has one public property, **ImageFrame**. The **ImageFrame.Image** member is a **PlanarImage** object that represents the image itself. The image is a single-plane RBG image with 32 bits per pixel.

The *nui_ColorFrameReady* method simply calls the WPF **BitmapSource.Create** method to create a bitmap for the display. It passes the following parameters:

- The width and height of the image, which are in the **Width** and **Height** members of the **PlanarImage** object.
- The horizontal and vertical resolution of the bitmap—96 dots per inch in each direction.
- The pixel format, identified by the **PixelFormats.Bgr32** property in the **System.Windows.Media** namespace.
- A null palette, because a palette is not used for this video image.
- The array of bits that form the image, from the **PlanarImage.Bits** member.
- The stride of the bitmap.

WPF handles display of the resulting bitmap.

Process Depth Data

When an image from the depth camera is ready, the runtime signals the **DepthFrameReady** event and calls *nui_DepthFrameReady*. This event handler retrieves the image frame, converts it to the desired display format, and calls the **BitmapSource.Create** method to display the resulting bitmap in the same way as for the video frame. The *nui_DepthFrameReady* method also updates the frames-per-second value on the screen. The following shows the code for this event handler:

```
void nui_DepthFrameReady(object sender, ImageFrameReadyEventArgs e)
{
  PlanarImage Image = e.ImageFrame.Image;
  byte[] convertedDepthFrame = convertDepthFrame(Image.Bits);

  depth.Source = BitmapSource.Create(Image.Width, Image.Height, 96, 96,
      PixelFormats.Bgr32, null, convertedDepthFrame, Image.Width * 4);

  ++totalFrames;

  DateTime cur = DateTime.Now;
  if (cur.Subtract(lastTime) > TimeSpan.FromSeconds(1))
  {
    int frameDiff = totalFrames - lastFrames;
    lastFrames = totalFrames;
    lastTime = cur;
    frameRate.Text = frameDiff.ToString() + " fps";
  }
}
```

When an application streams depth and player index, the depth data is an array of 16-bit values, as follows:

- The low-order 3 bits contain a skeleton ID.
- The remaining bits contain the depth value in millimeters.

If the application displayed the depth frame as raw, gray-scale data, users would find it difficult to distinguish the individuals in the scene. Therefore, the sample converts the 16-bit gray-scale data into 32-bit RGB data, with each person in a different color. The conversion routine operates on a byte array of depth data and steps through the array 2 bytes at a time. It follows these steps:

1. Store the player number from the low-order 3 bits of the first byte in the *player* variable. Values range from 0 to 6. A value equal to 0 indicates that no player is present.
2. Retrieve the 11 bits of actual depth data from the low-order 3 bits of the second byte and the high-order 5 bits of the first byte and store it in the *realDepth* variable.
3. Invert the result to generate an 8-bit intensity value that is appropriate for display, so that objects that are closer to the camera appear brighter and objects that are farther away appear darker. It ignores the most significant bit in this calculation.
4. Zero-initialize the corresponding elements of the 32-bit frame.
5. Assign red, green, and blue color values to the result based on the player number, by using the RED_IDX, GREEN_IDX, and BLUE_IDX constants to index into the array.

The following shows the code for the *convertDepthFrame* method:

```
byte[] convertDepthFrame(byte[] depthFrame16)
{
  for (int i16 = 0, i32 = 0;
      i16 < depthFrame16.Length && i32 < depthFrame32.Length;
      i16 += 2, i32 += 4)
  {
    int player = depthFrame16[i16] & 0x07;
    int realDepth = (depthFrame16[i16+1] << 5) | (depthFrame16[i16] >> 3);
    byte intensity = (byte)(255 - (255 * realDepth / 0x0fff));

    depthFrame32[i32 + RED_IDX] = 0;
    depthFrame32[i32 + GREEN_IDX] = 0;
    depthFrame32[i32 + BLUE_IDX] = 0;

    switch (player)
    {
      case 0:
         depthFrame32[i32 + RED_IDX] = (byte)(intensity / 2);
         depthFrame32[i32 + GREEN_IDX] = (byte)(intensity / 2);
         depthFrame32[i32 + BLUE_IDX] = (byte)(intensity / 2);
         break;
      case 1:
         depthFrame32[i32 + RED_IDX] = intensity;
         break;
      case 2:
         depthFrame32[i32 + GREEN_IDX] = intensity;
         break;
      case 3:
         depthFrame32[i32 + RED_IDX] = (byte)(intensity / 4);
         depthFrame32[i32 + GREEN_IDX] = (byte)(intensity);
         depthFrame32[i32 + BLUE_IDX] = (byte)(intensity);
         break;
      case 4:
         depthFrame32[i32 + RED_IDX] = (byte)(intensity);
         depthFrame32[i32 + GREEN_IDX] = (byte)(intensity);
         depthFrame32[i32 + BLUE_IDX] = (byte)(intensity / 4);
         break;
      case 5:
         depthFrame32[i32 + RED_IDX] = (byte)(intensity);
         depthFrame32[i32 + GREEN_IDX] = (byte)(intensity / 4);
         depthFrame32[i32 + BLUE_IDX] = (byte)(intensity);
         break;
     case 6:
         depthFrame32[i32 + RED_IDX] = (byte)(intensity / 2);
         depthFrame32[i32 + GREEN_IDX] = (byte)(intensity / 2);
         depthFrame32[i32 + BLUE_IDX] = (byte)(intensity);
         break;
```

```
    case 7:
        depthFrame32[i32 + RED_IDX] = (byte)(255 - intensity);
        depthFrame32[i32 + GREEN_IDX] = (byte)(255 - intensity);
        depthFrame32[i32 + BLUE_IDX] = (byte)(255 - intensity);
        break;
    }
  }
  return depthFrame32;
}
```

Process Skeleton Data

The *nui_SkeletonFrameReady* method handles **SkeletonFrameReady** events. This method retrieves a frame of skeleton data, clears any current skeletons from the display, and then draws line segments to represent the bones and joints of each tracked skeleton in the image, as follows:

```
void nui_SkeletonFrameReady(object sender, SkeletonFrameReadyEventArgs e)
{
  SkeletonFrame skeletonFrame = e.SkeletonFrame;
  int iSkeleton = 0;
  Brush[] brushes = new Brush[6];
  brushes[0] = new SolidColorBrush(Color.FromRgb(255, 0, 0));
  brushes[1] = new SolidColorBrush(Color.FromRgb(0, 255, 0));
  brushes[2] = new SolidColorBrush(Color.FromRgb(64, 255, 255));
  brushes[3] = new SolidColorBrush(Color.FromRgb(255, 255, 64));
  brushes[4] = new SolidColorBrush(Color.FromRgb(255, 64, 255));
  brushes[5] = new SolidColorBrush(Color.FromRgb(128, 128, 255));

  skeleton.Children.Clear();

  foreach (SkeletonData data in skeletonFrame.Skeletons)
  {
    if (SkeletonTrackingState.Tracked == data.TrackingState)
    {
      // Draw bones
      Brush brush = brushes[iSkeleton % brushes.Length];
      skeleton.Children.Add(getBodySegment(data.Joints, brush,
          JointID.HipCenter, JointID.Spine, JointID.ShoulderCenter,
          JointID.Head));
      skeleton.Children.Add(getBodySegment(data.Joints, brush,
          JointID.ShoulderCenter, JointID.ShoulderLeft, JointID.ElbowLeft,
          JointID.WristLeft, JointID.HandLeft));
      skeleton.Children.Add(getBodySegment(data.Joints, brush,
          JointID.ShoulderCenter, JointID.ShoulderRight, JointID.ElbowRight,
          JointID.WristRight, JointID.HandRight));
      skeleton.Children.Add(getBodySegment(data.Joints, brush,
          JointID.HipCenter, JointID.HipLeft, JointID.KneeLeft,
          JointID.AnkleLeft, JointID.FootLeft));
      skeleton.Children.Add(getBodySegment(data.Joints, brush,
          JointID.HipCenter, JointID.HipRight, JointID.KneeRight,
          JointID.AnkleRight, JointID.FootRight));
     // Draw joints
```

```
      foreach (Joint joint in data.Joints)
      {
        Point jointPos = getDisplayPosition(joint);
        Line jointLine = new Line();
        jointLine.X1 = jointPos.X - 3;
        jointLine.X2 = jointLine.X1 + 6;
        jointLine.Y1 = jointLine.Y2 = jointPos.Y;
        jointLine.Stroke = jointColors[joint.ID];
        jointLine.StrokeThickness = 6;
        skeleton.Children.Add(jointLine);
      }
    }
    iSkeleton++;
  }
}
```

The *nui_SkeletonFrameReady* method first retrieves the skeleton data from the *e* argument and stores it in a **SkeletonFrame** object. It then initializes the *iSkeleton* and *brushes* variables for use in drawing the bones of the skeleton. The *brushes* variable is an array of **System.Windows.Media.Brush** objects, and each element of the array is a **SolidColorBrush** object. Next, *nui_SkeletonFrameReady* clears any currently displayed skeleton by calling **Children.Clear** on the *skeleton* WPF canvas.

Now the method draws the skeleton: first the bones and then the joints.

The **skeletonFrame.Skeletons** member is an array of **SkeletonData** structures, each of which contains the data for a single skeleton. If the **TrackingState** field of the **SkeletonData** structure indicates that this skeleton is being tracked, *nui_SkeletonFrameReady* chooses a brush color and calls the *getBodySegment* method multiple times to draw lines that represent the connected bones of the skeleton.

The *getBodySegment* method has the following three parameters:

- A collection of joints for the skeleton, as a **Microsoft.Research.Kinect.Nui.JointsCollection** object.
- The brush with which to draw this line.
- A variable number of **JointID** values, where each value identifies a particular joint on the skeleton.

Most of the code in *getBodySegment* involves calls to **PointCollection** and **Polyline** methods in **System.Windows.Media**, so that code is not presented here. The *getBodySegment* method creates a collection of points and joins them with line segments. It returns a **Polyline** that connects the specified joints. The call to **skeleton.Children.Add** adds the returned line to the skeleton display.

The skeleton data, the color image data, and the depth data are based on different coordinate systems. To show consistent images from all three streams in the sample's display window, the program converts coordinates in skeleton space to image space by following these steps:

1. Convert skeleton coordinates in the [-1.0, 1.0] range to depth coordinates by calling **SkeletonEngine.SkeletonToDepthImage**. This function returns x and y coordinates as floating-point numbers in the range from 0.0 to 1.0.
2. Convert the floating-point coordinates to values in the 320x240 depth coordinate space, which is the range that **NuiCamera.GetColorPixelCoordinatesFromDepthPixel** requires.

3. Convert the depth coordinates to color image coordinates by calling **NuiCamera.GetColorPixelCoordinatesFromDepthPixel**. This method returns color image coordinates as values in the 640x480 color image space.
4. Scale the color image coordinates to the size of the skeleton display on the screen by dividing the x coordinate by 640 and the y coordinate by 480, and multiplying the results by the height and width of the skeleton display area, respectively.

The *getDisplayPosition* transformation function performs this work for each point, as follows:

```
private Point getDisplayPosition(Joint joint)
{
  float depthX, depthY;
  nui.SkeletonEngine.SkeletonToDepthImage(joint.Position,
      out depthX, out depthY);
  depthX = Math.Max(0, Math.Min(depthX * 320, 320));
  depthY = Math.Max(0, Math.Min(depthY * 240, 240));

  int colorX, colorY;
  ImageViewArea iv = new ImageViewArea();
  // only ImageResolution.Resolution640x480 is supported at this point
  nui.NuiCamera.GetColorPixelCoordinatesFromDepthPixel(
      ImageResolution.Resolution640x480, iv, (int)depthX, (int)depthY,
      (short)0, out colorX, out colorY);

  return new Point((int)(skeleton.Width * colorX / 640.0),
      (int)(skeleton.Height * colorY / 480));
}
```

When all bones have been drawn, *nui_SkeletonFrameReady* draws the joints. Each joint is drawn as a 6x6 box, in the color that was defined earlier in the *jointColors* dictionary. The *nui_SkeletonFrameReady* method transforms the starting (x,y) position of each joint by calling *getDisplayPosition*, just as *getBodySegment* transformed the bone positions.

WPF handles rendering of the resulting canvas on the display.

PART 4— Resources

The following links on the MSDN website provide additional information about Kinect video and related topics:

- Direct3D 9 Graphics
- Flipping Surfaces (Direct3D 9)
- System.Windows.Media Namespace
- Texture Filtering (Direct3D 9)
- Windows and Messages
- WinMain Entry Point
- WPF Windows Overview

For more information about implementing audio and related samples, see the Programming Guide page on the Kinect for Windows SDK Beta website at:

http://kinectforwindows.org